Tutorial Two: Working with Strings and Arrays

Imad Pasha
Chris Agostino

February 9, 2015

1 Some Experimentation in the Command Line

Before we get to writing up self contained programs, let's experiment around with making, editing, and combining lists, arrays, and strings in terminal.

1.1 Strings and Concatenation

The primary use for strings in a scientific code is for loading files with data. This is because a lot of the methods for loading data into python have functions which take as their argument the path location of the file, contained in a string. For example, we might have something like

```
np.loadtxt('/home/ipasha/simulation/data/file_1.txt')
```

to load a fits image. We aren't quite there yet, but we can get some practice with making strings and concatenating them together.

1. Open up terminal and launch the ipython interpreter.
2. Set a variable name equal to a string with your first and last name.
3. Print just your first name from the string by indexing it from the beginning to the last character of your first name.
4. Use string concatenation to set a variable full_name equal to the concatenation of your first name, (by indexing 'name'), a middle name (make one up if you don't have one), and your last name (by indexing name for just the characters in your last name). Make sure there are appropriate spaces between the words. You can do this in one line, or create intermediary variables and then concatenate them in a final line.

1.2 Arrays

Most of the data you will be dealing with will be stored (or you will want to store them) as numpy arrays. The focus of the program we write momentarily will be on arrays, but we are going to do a bit with them in the interpreter first to get a feel:

1. Import numpy as np, or as n (whichever you prefer, np is most commonly used)
2. Create an array of numbers 1 through 100. Print the last value of the array to make sure it worked (remember that by default, the first value it puts in an array is 0, and the final value is the number you select minus 1).
3. Set the array equal to itself times 2 and then print the full array.
4. Change the 5th element in the array to a 0.
5. Append any number to the end of the array (try looking up np.append( ) syntax.)
6. Print the maximum value, the minimum value, and the mean of the array (using numpy functions).
7. Reverse the array (see textbook).

2 Writing a Program

We will now move to arranging a coherent sequence of commands as a standalone program. Before we begin, we will briefly talk about what programs you can use to write and edit code.

2.1 VIM

Vim is a built in text editor in almost every terminal on a UNIX system (it even works in terminal on your Mac)! It has a slightly steeper learning curve than some other text editors, much like working in a terminal itself has a steeper learning curve than a GUI. However, once this is overcome, it is one of the most efficient ways of working with code. Like UNIX, vim has lots of short keyboard commands that make little to no sense. We will present the basics of opening, typing, and closing/saving here. See the VIM guide for more.

1. From terminal (in the directory you want the final file), type vim filename.py (you can use vim to make almost any file extension of text file, like .f90, .txt, etc.)
2. You will see a line of tildas. hit the ”i” key to enter insert mode, where you can type things in.
3. Once you have typed in the lines of code you want, hit the esc key to exit editing mode.
4. Use the command :wq to close and save the file. Typing :q! will close without saving any changes.

If you take the time to get comfortable with vim, you will be able to look at and edit code on any system with a terminal- whereas other text editing software may not be installed.

2.2 Other Programs

If you are using your own computer or laptop to write code, there are plenty of options for code text editors. You should already have canopy installed as your python distribution. Opening the Canopy software should allow you to edit and save (and run) .py files. Sublime and notepad++ are two other popular choices. These have familiar gui interfaces for saving and editing text.

3 Looking at Spectra

Astronomers basically have one way of learning about the universe: detecting photons. Pretty much everything we know in astrophysics comes from the analysis of data which is nothing more than photon counts. In addition to images (such as those taken by Hubble), Astronomers can take spectra of light sources to determine their elemental composition, and sometimes also the studied object's velocity. A spectrum is basically what you see when you use a prism to spread out white light. Advanced spectrometers use what is known as an echelle grating to split up light and "bin" it by energy, or wavelength, since the two are the same, with some constant multipliers (fig. 1). Because of it's importance, 3 out of four of the Astro 120 labs typically involve some sort of spectroscopy, or analysis of spectra.

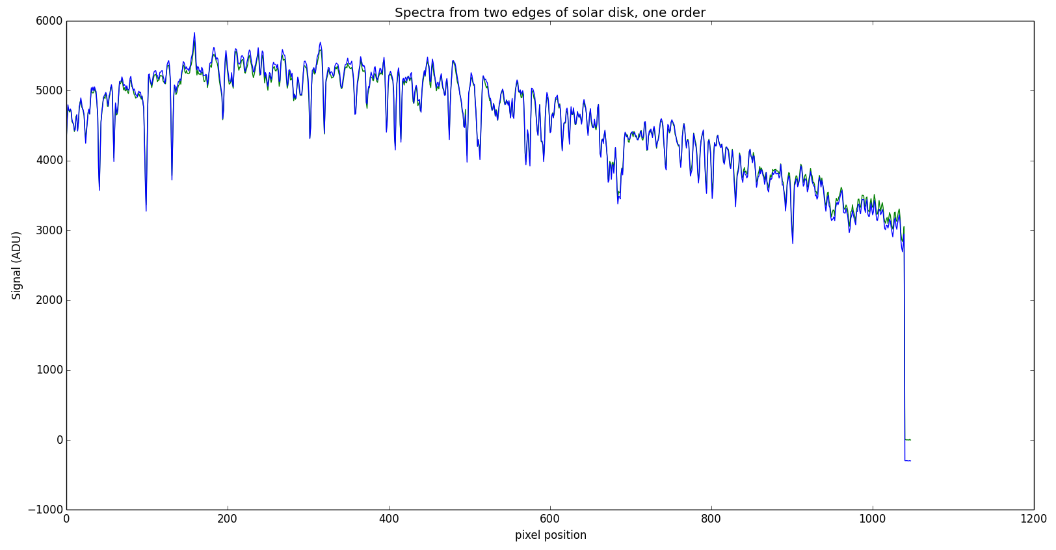

Figure 1: An example of a typical Solar Spectrum. The characteristic "dips" in the continuum spectrum are known as absorbtion lines. They occur when photons passing through the outer atmosphere of the sun are absorbed by stellar gas and then emitted at random directions (and possibly different wavelenths), resulting in a "dip" in the brightness of the light along our line of sight at that wavelength.

Over the course of this class, we are going to be spending some time building up an understanding for how to work with spectra, (a) because the upper division lab requires it over and over, and (b) because spectra are stored in arrays, and are a good way of developing a feel for array manipulation while having some physical motivation. Today, will start with writing a program to load up a spectrum file, make some corrections and modifications to the data, and then plot and view it.

3.1 Writing the Script

Let's get down to writing the script to look at some spectra. Open/create a file called spectrum_load.py and import numpy and matplotlib at the top. We will be using matplotlib a bit, even though we haven't covered the details of matplotlib yet, but don't worry. Plotting steps will be spelled out, and we are only using the simplest plotting functions.

1. From the tutorial page you can download the neon spectrum file – use the syntax mentioned in section 1.1 to load up that file and store it in a variable called spectrum. As of now, the data are "pairs" of numbers, (pixel,value), but we want the full array of pixels, and the full array of intensities separately (for plotting purposes). Do this by setting spectrum = np.transpose(spectrum).
    - There were two columns in the data file, pixel number (the independent), and signal strength/intensity/brightness (dependent). They are now both stored in "spectrum" and can be accessed by indexing spectrum[0] and spectrum[1].
2. Make two new variables, called "pixels" and "signal", and set them equal to the proper index of "spectrum". If this step is confusing, call one of us over to explain. What you should end up with is one array that contains all of the wavelengths, and one with all of the intensity values.
3. We will now perform some simple adjustments. Sometimes, we bring in data that is reversed- say the arrays you just imported would be showing decreasing wavelength rather than increasing. That is the case here, so add a line of code to reverse the array.
4. Sometimes there are leading and trailing tails of data that were taken before an instrument turned on, etc. To exclude them, we perform a step known as "truncation." In this data set, the first 500 values are not data we need to use. Use array indexing to truncate the two arrays, pixels and signal, so that the first 500 values are **not** included in the arrays.
    - You can choose to make new variables like truncated_pix and truncated_signal, or just change the original arrays. In practice, the first way is safer and easier to bug-fix, while the second way is faster and doesn't introduce a bunch of new variables to remember. We choose the second route, so it may be easier to follow along if you do as well.
5. Sometimes we want to compare multiple spectra, but for various reasons the intensities are on different scales (the light source may have gotten brighter, for example). Often a step we take is normalizing the data, to allow for comparison. Normalize the array with the signal values by setting a variable "normed_signal" equal to "truncated_signal" divided by its mean value.

- This normalizes your data around a mean of 1. There are many other normalizations possible. Also, note that this step takes away information you have about the raw signal levels in each spectrum. That's why this warrants a new variable, "normed_signal" to store the normalized spectrum separately from the original. Other popular normalizations include dividing by the maximum of the array (leaving an array with a peak value at one).

Now that we have taken the steps to prep our data to be visualized, we can go ahead and graph it. Since we haven't covered this yet, we will tell you the code you need. Try to figure out what each line does, and we will discuss it more next week.

Enter the following code at the end of your document you are working on (this assumes matplotlib.pyplot is imported as plt:

```
plt.plot(truncated_pix, normed_signal)
plt.title("A Neon Spectrum")
plt.xlabel("Pixel Number")
plt.ylabel("Signal [adu]")
plt.show()
```

*Note: of course, if you had different variable names for the final, prepped pixel and signal arrays you would plug those in. Just to give some motivation for what's to come: notice how in its current state we can only open one spectrum at a time. Through the power of looping and conditionals, we can actually load up many files at the same time to work with.

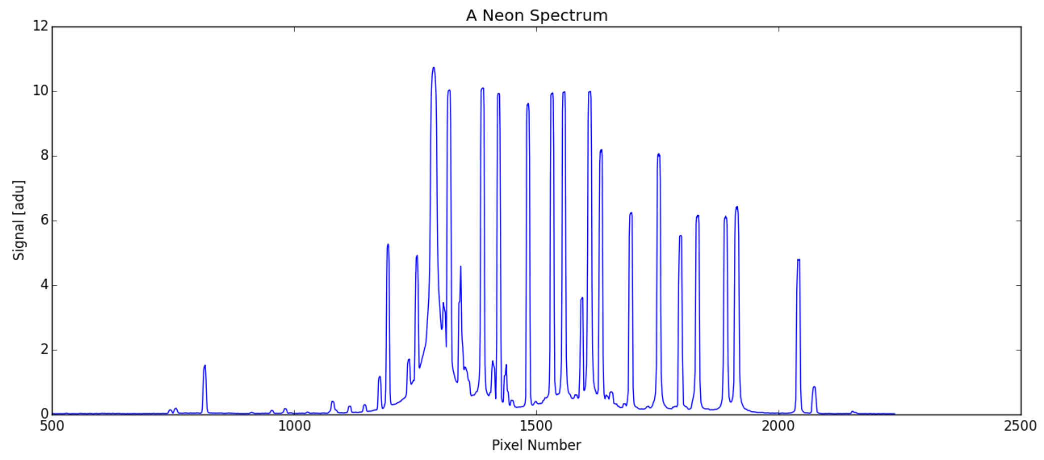

Figure 2: The final Neon spectrum.

3.2 Saving and running

Save your file, and then open the ipython interpreter in the same directory as your file (if you are working in canopy you won't need to do this). Run your file, and see if python pops up a spectrum that looks like fig. 2. If it does, congrats! You're done with the tutorial for the week. You'll be turning in your python script, plot, and homework script (below)! Make sure your code is commented, and please zip or tar the 3 files into one .tar, .zip, or .rar before submitting, so there's not 3x41 files floating around! Also, when you save the .tar or .zip file, make sure your name is in the filename, along with the tutorial number.

4 Homework

A script to solve the quadratic equation
Your homework for this week is to create a python program which solves the quadratic formula based on user inputs. Let's start with the quadratic formula. We all know it to be

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

We need people to be able to enter the coefficients of their quadratic into our program. For that, we use the function raw_input(). The way this function works is as follows:

```
a = raw_input('Enter the leading coefficient: ')
```

where the text in the parenthesis will appear to screen, and wait for the user to enter something before storing it in a string called a. (You will have to recast a as a float before using it in the final calculation). Using the raw_input function, have the user enter the a, b, and c terms from their polynomial they want to solve and turn them into floats. Now that we have the three terms, we can have a line that sets x = the formula above. It would use a lot of parenthesis, so if you want, you could separately calculate for the discriminant, denominator, etc., and then perform the calculations with those variables, i.e.,

```
x = (-b + np.sqrt(discriminant)) / denominator
```

*Note that you will need to set something like x1 equal to the plus term and x2 = to the minus term. So now we should have the solutions to the quadratic equation. We need to be able to print the results to the screen so the user can see them. The simplest way to do so would be

```
print x1, x2
```

but we want to be a bit fancier than that. Instead we could write

```
print 'The x values which solve this quadratic are: ', x1, x2
```

and python will print out the sentence with the answers in it.

And that's it! You can turn in your script with the code for your quadratic solver along with your tutorial and neon plot to the website.

Please read the textbook up through the section on file-writing (So including function writing and conditionals). Go over any earlier section that you may have had questions about. If you haven't read chapter 5 yet, do so, because there will be more plotting in the next tutorial!

5 Bonus Homework

This section is for anyone who breezed through the tutorial/homework and wants more to do, or for anyone that really wants to push themselves with slightly more advanced material.

Notice how our quadratic solver assumes the entered values of a, b, and c produce a positive discriminant? If we try to plug in values to a quadratic which has no real roots, python will throw an error that it tried to square root a negative number. Let's see if we can address that possibility using if-statements (which will be covered next week) and return a error message if they try to solve a polynomial without real roots.

The basics of an if statement is it will check if some statement is true- if it is, it will run one block of code, if it is not, it will run another. What you need to do is put an if statement before your calculation of x1 and x2 (but after you set the variables discriminant, denominator, etc), that looks like this:

if (discriminant > 0) or (discriminant == 0):

after which you need to indent the rest of your code (with the calculations and print statements) by one indent. (This is because we want that entire "block" of code to run only if that if statement is true).

In its current state, if you entered numbers that produced a positive or 0 discriminant, the code would run normally, but if it were negative, it would skip the rest of your code and nothing would happen. Now we can address the negative case. Underneath your last line of code (the indented print statement), put another line (NOT indented) that reads:

elif discriminant < 0:

Indent yourself one tab, and you are now ready to write the block of code to handle what should happen with a negative discriminant. FYI, the elif just stands for "else, if". Within this new indented block of code, introduce a print statement to tell the user that they entered a polynomial without real roots.